Capitolo 2

Fondamenti del linguaggio Java

di Michael Morrison

IN QUESTO CAPITOLO

Java è un linguaggio orientato agli oggetti, dunque si basa sul concetto di *oggetto*. In ogni caso, benché la conoscenza della programmazione orientata agli oggetti sia necessaria per utilizzare Java, non è richiesta per comprendere i fondamenti del linguaggio. Questo capitolo si concentra sul linguaggio, mentre i dettagli della programmazione orientata agli oggetti sono presentati nel Capitolo 4.

Java può quasi essere considerato una forma rinnovata del C++, e nei prossimi capitoli vengono evidenziate molte delle analogie e differenze con questo linguaggio.

Ciao, mondo!

Il modo migliore per apprendere un linguaggio di programmazione è quello di vedere subito come funziona un programma vero; per rispettare la tradizione, nel Listato 2.1 è riportato il codice del classico programma "Ciao, mondo!", disponibile anche nel CD allegato come file `HelloWorld.java`.

Listato 2.1 *La classe CiaoMondo.*

```
Class CiaoMondo {
  public static void main(String args[]) {
    System.out.println("Ciao, mondo!");
  }
}
```

Dopo la compilazione con `javac`, il programma può essere eseguito con l'interprete `java`. Il file eseguibile creato dal compilatore è denominato `CiaoMondo.class`. Mentre nella maggior parte dei linguaggi di programmazione si utilizza l'estensione `.exe` per gli eseguibili, in Java non è così. Per eseguire il programma `CiaoMondo` occorre digitare **java CiaoMondo** al prompt di comandi: appare così il messaggio: "Ciao, mondo!" sullo schermo.

Naturalmente questo è un programma ridotto all'osso, tuttavia presenta diversi aspetti interessanti. Per prima cosa occorre comprendere che Java si basa pesantemente sulle classi, e in effetti la prima istruzione del programma fa capire che `CiaoMondo` è una classe e non un programma. Inoltre, il nome della classe è utilizzato dal compilatore come nome del file di output eseguibile. Se un file sorgente `.java` contiene diverse definizioni di classe, il compilatore le inserisce ognuna in un file `.class` separato.

La classe `CiaoMondo` contiene un unico *metodo*, o funzione membro. Per ora, si possono considerare i metodi come normali funzioni procedurali collegate alla classe; per ulteriori dettagli si rimanda al Capitolo 4. Il metodo di questo esempio si chiama `main()` e dovrebbe risultare familiare a chi proviene dal C/C++. Si tratta del metodo dove inizia l'esecuzione del programma; è definito come `public static` e restituisce `void`. `public` significa che il metodo può essere richiamato dall'interno o dall'esterno della classe, `static` che il metodo è identico per tutte le istanze della classe; `void` significa che il metodo non restituisce alcun valore.

Il metodo `main()` accetta un unico parametro, `String args[]`. `args[]` è un array di oggetti `String` che rappresentano argomenti della riga di comando passati alla classe al momento dell'esecuzione. Poiché `CiaoMondo` non utilizza alcun argomento della riga di comando, si può ignorare questo parametro.

Il metodo `main()` è richiamato all'esecuzione della classe `CiaoMondo`; è costituito da una singola istruzione che invia il messaggio: "Ciao, mondo!" sul dispositivo di output standard:

```
System.out.println("Ciao, mondo!");
```

L'istruzione può risultare complicata per l'annidamento degli oggetti. Esaminandola da destra a sinistra, si nota innanzitutto che termina con un punto e virgola, infatti la sintassi standard di Java è ripresa da quella del C/C++. Andando verso sinistra, si vede che la stringa "Ciao, mondo!" è racchiusa tra parentesi, per indicare che si tratta di un parametro per una funzione. Il metodo richiamato è `println()`, un metodo dell'oggetto `out`. Questo metodo è simile al metodo `printf()` del C, eccetto per il fatto che aggiunge automaticamente un'interruzione di riga (\n) alla fine della stringa.

L'oggetto `out` è una variabile membro dell'oggetto `System`, che rappresenta il dispositivo di output standard. Infine, l'oggetto `System` è un oggetto globale dell'ambiente di Java che incapsula funzionalità di sistema.

Il resto del capitolo è dedicato all'esame dei fondamenti del linguaggio.

Token

Quando si invia un programma Java al compilatore, questo analizza il testo ed estrae i singoli *token*, ovvero i più piccoli elementi di programma comprensibili per il compilatore. I token definiscono la struttura del linguaggio Java e possono essere suddivisi in cinque categorie: identificatori, parole chiave, letterali, operatori e separatori. Il compilatore inoltre riconosce ed elimina commenti e spazi bianchi.

I token risultanti dall'analisi del codice sono poi compilati in bytecode indipendente dalla macchina, che può essere eseguito da qualsiasi interprete Java. Il bytecode è conforme all'ipotetica macchina virtuale di Java, che astrae le differenze tra i processori in un unico processore virtuale.

Identificatori

Gli *identificatori* sono token che rappresentano nomi. Questi nomi possono essere assegnati a variabili, metodi e classi per identificarli in modo univoco per il compilatore e fornirne una descrizione per il programmatore. `CiaoMondo` è un identificatore che assegna il nome `CiaoMondo` alla classe che si trova nel file sorgente `CiaoMondo.java`.

Per i nomi degli identificatori in Java esistono delle limitazioni: tutti distinguono tra maiuscole e minuscole e devono iniziare con una lettera, un trattino di sottolineatura (_) o un segno di dollaro ($). La lettera iniziale può essere maiuscola o minuscola, tra gli altri caratteri vi possono essere le cifre da 0 a 9. L'unica altra limitazione è l'impossibilità di utilizzare le parole chiave di Java, elencate nel paragrafo successivo. Nella Tabella 2.1 sono elencati alcuni esempi di nomi validi e non validi come identificatori.

Tabella 2.1 *Esempi di identificatori validi e non validi in Java.*

Valido	*Non valido*
`CiaoMondo`	`Ciao Mondo` (c'è uno spazio).
`Ciao_Mamma`	`Ciao Mamma!` (c'è uno spazio e un segno di punteggiatura).
`CiaoBello2`	`3CiaoBello` (inizia con un numero).
`corto`	`short` (è una parola chiave di Java).
`percentuale`	`%percento` (non inizia con una lettera).

Oltre alle restrizioni sui nomi degli identificatori, esistono regole stilistiche per migliorare la leggibilità e la coerenza del codice. Gli identificatori a più parole sono riportati in minuscolo, con l'eccezione della lettera iniziale di parole che si trovano all'interno del nome. Ad esempio, il nome `bellaDonna` è nello stile corretto, mentre `belladonna`, `BellaDonna` e `BELLADONNA` non lo sono.

Inoltre, si sconsiglia l'utilizzo del trattino di sottolineatura e del segno di dollaro come iniziale, perché in molte librerie del C si utilizzano nomi di questo tipo. Il trattino è utile per separare parole dove normalmente si inserirebbe uno spazio, come in `Ciao_Mamma`.

Con Java 1.1 le convenzioni di stile per i nomi hanno assunto un'importanza ancora maggiore, perché da esse dipende in parte il funzionamento di JavaBeans. Ulteriori informazioni su JavaBeans si trovano nella Parte 8.

Parole chiave

Le parole chiave sono identificatori predefiniti utilizzati soltanto in modo specifico e limitato. Eccone l'elenco:

```
abstract   double      int        super
boolean    else        interface  switch
break      extends     long       synchronized
byte       false       native     this
byvalue    final       new        threadsafe
case       finally     null       throw
catch      float       package    transient
char       for         private    true
class      goto        protected  try
const      if          public     void
continue   implements  return     while
default    import      short
do         instanceof  static
```

Letterali

Gli elementi di programma utilizzati sempre nello stesso modo sono chiamati *letterali*, o *costanti*, e possono essere numeri, caratteri o stringhe. I letterali numerici possono essere interi, numeri in virgola mobile e booleani (per un retaggio del C, dove i valori booleani per vero e falso sono 1 e 0). I letterali carattere si riferiscono sempre a un singolo carattere Unicode. Le stringhe, che contengono più caratteri, sono considerate letterali anche se in Java sono implementate come oggetti.

Il set di caratteri Unicode è un set a 16 bit che costituisce il set di caratteri ASCII. I 16 bit consentono di rappresentare molti simboli e caratteri di altre lingue. Unicode sta rapidamente imponendosi come standard per i moderni sistemi operativi.

Letterali interi

I *letterali interi* sono quelli utilizzati più spesso in Java: possono essere decimali, esadecimali e ottali; questi formati corrispondono alla base del sistema di numerazione utilizzato. I letterali decimali (base 10) appaiono come numeri normali senza una notazione particolare, quelli esadecimali (base 16) appaiono con `0x` oppure `0X` in testa, come in C/C++, quelli ottali (base 8) appaiono con `0` in testa. Ad esempio, il letterale intero decimale 12 è rappresentato come `12` in formato decimale, `0xC` in formato esadecimale e `014` in formato ottale.

I letterali interi per default sono memorizzati nel tipo `int`, un valore con segno a 32 bit. Per numeri più grandi si può utilizzare il tipo `long` a 64 bit, aggiungendo `l` o `L` alla fine del numero, come in `79L`.

Letterali in virgola mobile

I letterali in virgola mobile rappresentano numeri decimali con una parte frazionaria, come 3.142 (naturalmente si utilizza il punto decimale). Possono essere espressi in notazione standard o scientifica; ad esempio, il numero 563.84 può essere espresso come 5.6384e2.

A differenza dei letterali interi, quelli per in virgola mobile per default sono del tipo `double`, a 64 bit. Si può scegliere di utilizzare il formato a 32 bit `float` se opportuno, aggiungendo `f` o `F` alla fine, come in 5.6384e2F.

Letterali booleani

I *letterali booleani* sono una novità rispetto al C/C++, dove i valori vero e falso sono rappresentati da 1 e 0. In Java invece esistono i due stati `true` e `false`.

I letterali booleani in Java sono utilizzati quasi quanto gli interi, poiché sono presenti in quasi tutti i tipi di strutture di controllo, ovunque occorra rappresentare una condizione o uno stato con due valori possibili.

Letterali carattere e caratteri speciali

I *letterali carattere* rappresentano un singolo carattere Unicode e appaiono racchiusi tra apici singoli. Come in C/C++, i *caratteri speciali* (caratteri di controllo e non stampabili) sono rappresentati da una barra inversa (\) seguita dal codice del carattere, come `\n`, che forza un'interruzione di riga. Nella Tabella 2.2 sono elencati i caratteri speciali supportati da Java.

Un esempio di letterale carattere Unicode è `\u048`, che è una rappresentazione esadecimale del carattere H. Lo stesso carattere è rappresentato in ottale da `\110`.

Per ulteriori informazioni sul set di caratteri Unicode si rimanda al sito `http://www.unicode.org`.

Tabella 2.2 *Caratteri speciali supportati da Java.*

Descrizione	*Rappresentazione*
Barra inversa	`\\`
Continua	`\`
Ritcanc	`\b`
Ritorno carrello	`\r`
Avanzamento pagina	`\f`
Tabulazione orizzontale	`\t`
Nuova riga	`\n`
Apice singolo	`\'`
Apice doppio	`\"`
Carattere Unicode	`\u`*`ddd`*
Carattere ottale	`\`*`ddd`*

Letterali stringa

I *letterali stringa* rappresentano caratteri multipli e appaiono racchiusi tra apici doppi. A differenza degli altri letterali, questi sono implementati dalla classe `String`, diversamente da quanto avviene in C/C++, dove le stringhe sono considerate array di caratteri.

Operatori

Gli *operatori* indicano un calcolo da eseguire un uno o più dati, chiamati *operandi*, che possono essere letterali, variabili o risultati di funzioni. Ecco gli operatori supportati da Java:

```
+      -      *      /      %      &      |
^      ~      &&     ||     !      <      >
<=     >=     <<     >>     >>>    =      ?
++     --     ==     +=     -=     *=     /=
%=     &=     |=     ^=     !=     <<=    >>=
>>>=   .      [      ]      (      )
```

Ulteriori informazioni al riguardo sono riportate nel Capitolo 3.

Separatori

I *separatori* sono utilizzati per indicare al compilatore come sono raggruppati gli elementi nel codice. Ad esempio, gli elementi di un elenco sono separati da virgole. Ecco i separatori supportati da Java:

```
{   }   ;   ,   :
```

Commenti e spazi bianchi

In precedenza si è detto che commenti e spazi bianchi sono rimossi dal compilatore durante l'analisi del codice sorgente. Gli spazi bianchi possono essere spazi, tabulazioni e caratteri di nuova riga. I commenti possono essere definiti in tre modi, descritti nella Tabella 2.3.

Tabella 2.3 *Tipi di commenti supportati da Java.*

Tipo	*Utilizzo*
`/* commento */`	Tutti i caratteri compresi tra /* e */ sono ignorati.
`// commento`	Tutti i caratteri dopo // fino alla fine della riga sono ignorati.
`/** commento */`	Identico a /* */, ma questa forma può essere utilizzata con `javadoc` per creare automaticamente la documentazione.

Ecco alcuni esempi di commenti.

```
/* Questo è un commento nello stile del C. */
// Questo è un commento nello stile del C++.
/** Questo è un commento adatto per javadoc. */
```

Tipi di dati

Uno dei concetti fondamentali di qualsiasi linguaggio di programmazione è rappresentato dai *tipi di dati*, che definiscono i metodi di memorizzazione dei dati disponibili per rappresentare le informazioni e il modo in cui queste sono interpretate. I tipi di dati sono strettamente connessi alla memorizzazione delle variabili, infatti determinano il modo in cui il compilatore interpreta il contenuto della memoria.

Per creare una variabile in memoria, occorre dichiararla specificandone il tipo, oltre a un identificatore. La sintassi utilizzata in Java è la seguente:

```
Tipo Identificatore [, Identificatore];
```

L'istruzione di dichiarazione indica al compilatore di riservare della memoria per una variabile di tipo `Tipo` con il nome `Identificatore`. Le parentesi indicano che è possibile effettuare più dichiarazioni dello stesso tipo sulla stessa riga, separandole con virgole. Infine, come tutte le istruzioni di Java, anche quella di dichiarazione termina con un punto e virgola.

I tipi di dati in Java si suddividono in due categorie: semplici e composti. I tipi di dati semplici sono tipi di base non derivati da altri, come interi, in virgola mobile, booleani e carattere. I tipi composti, array, stringhe, classi e interfacce, si basano su quelli semplici.

I tipi di dati interi

I tipi di dati interi sono utilizzati per rappresentare numeri interi con segno. Esistono quattro tipi interi: `byte`, `short`, `int` e `long`, ognuno dei quali richiede quantità di memoria diverse (Tabella 2.4).

Tabella 2.4 *Tipi interi di Java.*

Tipo	*Dimensione*
`byte`	8 bit
`short`	16 bit
`int`	32 bit
`long`	64 bit

Per dichiarare variabili di tipo intero, si utilizza la sintassi descritta in precedenza con il tipo desiderato. Ecco alcuni esempi:

```
int i;
short provaLivello;
long angolo, ampiezza;
bute rosso, verde, blu;
```

Tipi di dati in virgola mobile

I tipi di dati in virgola mobile sono utilizzati per rappresentare numeri con parti frazionarie; ne esistono due: `float` e `double`. Il tipo `float` memorizza numeri a singola precisione in 32 bit, `double` numeri a doppia precisione in 64 bit.

La dichiarazione è analoga a quella degli altri tipi. Ecco alcuni esempi:

```
float temperatura;
double velocitaVento, pressioneBarometrica;
```

Tipo di dati booleano

Il tipo di dati booleano (`boolean`) è utilizzato per memorizzare valori con due stati possibili: `true` o `false`. Può essere considerato come un valore intero a 1 bit (poiché 1 bit può assumere solo due valori: 1 o 0). Le parole chiave `true` e `false` sono i soli valori booleani validi (a differenza del C/C++). Ecco un esempio di dichiarazione:

```
boolean giocoFinito;
```

Tipo di dati carattere

Il tipo di dati carattere (`char`) è utilizzato per singoli caratteri Unicode. Poiché il set Unicode è composto di valori a 16 bit, il tipo carattere è memorizzato come intero a 16 bit senza segno; ecco un esempio di dichiarazione:

```
char inizialeNome, inizialeCognome;
```

In C e C++ le stringhe sono formate da array di caratteri, mentre in Java si utilizza la classe `String`. Ciò non significa che in Java non si possano creare array di caratteri, ma occorre fare attenzione a non utilizzare un carattere quando in realtà serve una stringa: in C e C++ non vi è distinzione, in Java sì.

Casting di tipi di dati

Il processo di conversione da un tipo di dati all'altro si chiama *casting* ed è spesso necessario quando una funzione restituisce un tipo diverso da quello richiesto per eseguire un'operazione. Ad esempio, la funzione membro `read()` del dispositivo di input standard (`System.in`) restituisce un `int`, di cui occorre effettuare il casting in `char` prima di memorizzarlo, come segue:

```
char c = (char)System.in.read();
```

Il casting si esegue inserendo il tipo desiderato tra parentesi a sinistra del valore da convertire. La chiamata a `System.in.read()` restituisce un valore `int`, che viene convertito in `char` da `(char)`. Il risultato è infine memorizzato in `c`.

La dimensione in memoria dei tipi di cui si effettua il casting è molto importante, infatti non sempre è possibile eseguire il casting da un tipo all'altro. Si consideri il casting di un `long` in un `int`: il primo è un valore a 64 bit, il secondo a 32, perciò il compilatore elimina i primi 32 bit del valore `long`, e quindi la relativa informazione va perduta nel casting. Lo stesso problema si verifica quando si esegue il casting tra due tipi semplici, come interi e valori in virgola mobile. Ad esempio, il casting di un `double` in un `long` causa la perdita della parte frazionaria, anche se entrambi i valori sono a 64 bit.

Nella Tabella 2.5 sono elencati i casting che non causano perdite di informazioni.

Tabella 2.5 *Casting senza perdite di informazioni.*

Dal tipo	*Al tipo*
`byte`	`short, char, int, long, float, double`
`short`	`int, long, float, double`
`char`	`int, long, float, double`
`int`	`long, float, double`
`long`	`float, double`
`float`	`double`

Blocchi e ambito

In Java, il codice sorgente è diviso in porzioni separate da parentesi graffe aperte e chiuse { e }; tutto quanto è compreso tra una coppia di parentesi graffe è considerato un *blocco* ed è più o meno indipendente da quanto si trova al di fuori. I blocchi sono importanti non solo dal punto di vista logico, ma anche per la sintassi, poiché se non si utilizzano i blocchi, il compilatore non riesce a determinare dove termina una sezione di codice e inizia la successiva. Tra l'altro, essi sono praticamente indispensabili per una buona leggibilità del codice.

Tutto quanto fa parte di un blocco può essere considerato come un'unica istruzione. In realtà un blocco è semplicemente una parte di codice. I blocchi possono essere annidati gli uni dentro gli altri in modo gerarchico.

Nello stile di programmazione standard, i blocchi sono identificati tramite dei rientri; ogni volta che si inserisce un nuovo blocco, occorre rientrare il codice di un certo numero di spazi, solitamente 2, e quando il blocco termina si torna a sinistra di due spazi. L'utilizzo dei rientri, in ogni caso, non è obbligatorio dal punto di vista tecnico: l'output del compilatore è identico anche senza rientri, comunque è consigliabile per la leggibilità. Ecco un esempio di buon utilizzo dei rientri:

```
for (int i = 0; i < 5; i++) {
  if (i < 3) {
    System.out.println(i);
  }
}
```

Ed ecco lo stesso codice senza i rientri:

```
for (int i = 0; i < 5; i++) {
if (i < 3) {
System.out.println(i);
}
}
```

Il primo esempio è decisamente più chiaro: risulta evidente che l'istruzione `if` è annidata all'interno del ciclo `for`. Il secondo esempio invece non è così leggibile.

Il concetto di *ambito* è strettamente legato ai blocchi ed è molto importante quando si lavora con le variabili; esso indica la porzione di codice in cui una variabile "vive", o risulta visibile e disponibile. Ogni variabile ha un ambito, quindi è utilizzata in una sezione particolare.

L'ambito è determinato dai blocchi. Per comprendere meglio, si consideri ancora il Listato 2.1 riportato in precedenza: la classe `CiaoMondo` è composta da due blocchi. Il blocco esterno è quello che definisce la classe:

```
class CiaoMondo {
...
}
```

I blocchi di classe sono molto importanti, poiché ad esempio i metodi sono definiti all'interno delle classi a cui appartengono. Sintatticamente e logicamente, tutto in Java avviene all'interno di una classe. Tornando a `CiaoMondo`, il blocco interno definisce il codice del metodo `main()`:

```
public static void main(String args[]) {
  ...
  }
```

Questo blocco è considerato annidato all'interno di quello esterno. Le variabili definite nel blocco interno sono locali a tale blocco e non sono visibili al blocco esterno. In questo caso l'ambito è definito come il blocco interno.

Per comprendere meglio il significato di ambito e blocchi, si consideri la classe `SalveMondo`, riportata nel Listato 2.2.

Listato 2.2 *La classe SalveMondo.*

```
class SalveMondo {
  public static void main (String args[]) {
    int i;
    stampaMessaggio();
  }
  public static void stampaMessaggio () {
    int j;
    System.out.println("Salve, mondo!");
  }
}
```

Questa classe contiene due metodi, `main()` e `stampaMessaggio()`. Il primo è simile a quello della classe `CiaoMondo`, ma dichiara una variabile `i` e richiama il metodo `stampaMessaggio()`, che a sua volta dichiara una variabile `j` e invia il messaggio: "Salve, mondo!" al dispositivo di output standard.

È facile indovinare che il risultato di `SalveMondo` è del tutto analogo a quello di `CiaoMondo`, perché la chiamata di `stampaMessaggio()` causa la visualizzazione di una frase di testo.

Meno evidente è l'ambito delle variabili definite nei due metodi. L'intero `i` definito in `main()` ha un ambito limitato al corpo del metodo `main()`, definito dal blocco corrispondente (delimitato dalle parentesi graffe). Similmente, la variabile `j` ha un ambito limitato al corpo del metodo `stampaMessaggio()`. Le variabili non sono visibili al di fuori del proprio ambito, e perciò non sono visibili al blocco della classe `SalveMondo`. Inoltre, `j` non è visibile al metodo `main()` e `i` non è visibile al metodo `stampaMessaggio()`.

L'ambito assume maggiore importanza quando si annidano blocchi di codice gli uni dentro gli altri. La classe `AddioMondo` del Listato 2.3 offre un buon esempio.

Listato 2.3 *La classe AddioMondo.*

```
class AddioMondo {
  public static void main (String args[]) {
    int i, j;
    System.out.println("Addio, mondo!");
    for (i = 0; i < 5; i++) {
      int k;
      System.out.println("Addio!");
    }
  }
}
```

Gli interi i e j hanno un ambito limitato al corpo del metodo main(), mentre l'intero k ha un ambito limitato al blocco del ciclo for e quindi non è visibile al di fuori di esso. D'altra parte, i e j rimangono visibili all'interno del blocco del ciclo for. L'ambito infatti funziona in modo gerarchico dall'alto verso il basso: le variabili definite in blocchi più esterni sono visibili anche nei blocchi interni, mentre non vale il contrario.

È importante prestare attenzione all'ambito delle variabili al momento della dichiarazione, e non solo per la visibilità, poiché l'ambito determina anche la *vita*. Infatti le variabili vengono distrutte quando escono dal proprio ambito. Nel Listato 2.3, la memoria per gli interi i e j è allocata quando il programma arriva al metodo main(), e la memoria per l'intero k è allocata quando si entra nel ciclo for. Quando l'esecuzione abbandona il blocco del ciclo for, la memoria di k è liberata e la variabile distrutta; lo stesso avviene per i e j quando l'esecuzione abbandona main(). Ambito e vita assumono importanza ancora maggiore quando si ha a che fare con le classi, come viene spiegato nel Capitolo 4.

Array

Un *array* è un costrutto che fornisce lo spazio per un elenco di elementi dello stesso tipo. Gli elementi di un array possono essere di tipo semplice o composto. Gli array possono anche essere multidimensionali. In Java gli array sono dichiarati con una coppia di parentesi quadre [], come nei seguenti esempi:

```
int numeri[];
char[] lettere;
long griglia[][];
```

Chi conosce già gli array potrebbe rimanere sorpreso dalla mancanza di numeri all'interno delle parentesi per specificare il numero di elementi dell'array. Java non consente di specificare la dimensione di un array vuoto al momento della dichiarazione, occorre utilizzare allo scopo l'operatore new o assegnare un elenco di elementi al momento della creazione. L'operatore new è trattato nel capitolo seguente.

Il motivo per cui occorre utilizzare l'operatore new *per impostare la dimensione degli array è che Java non dispone dei puntatori come C e C++, perciò non consente di puntare all'interno degli array per creare nuovi elementi. In questo modo si evitano però i problemi di controllo dei limiti comuni in C e C++.*

Un'altra stranezza è la possibilità di disporre le parentesi quadre dopo il tipo o dopo l'identificatore della variabile. Ecco due esempi di array dichiarati e impostati a una dimensione specifica con l'operatore new e assegnando un elenco di elementi nella dichiarazione:

```
char alfabeto[] = new char[26];
int primi = {7, 11, 13};
```

Java supporta anche strutture più complesse per memorizzare elenchi di elementi, come *stack* e *tabelle di hash*, che però sono implementati come classi e trattati nel Capitolo 11.

Stringhe

In Java le *stringhe* sono gestite da una classe speciale denominata `String`. Anche le stringhe letterali sono gestite come istanze di questa classe. Un'*istanza* di una classe è semplicemente un oggetto creato in base alla descrizione della classe. Questo metodo di gestire le stringhe è molto diverso da quello di linguaggi come C e C++, dove le stringhe sono rappresentate semplicemente come array di caratteri. Ecco alcune stringhe dichiarate con la classe `String` di Java:

```
String messaggio;
String nome = "Sig. Biondo";
```

A questo punto non è necessario conoscere nei dettagli la classe `String`, che viene esaminata nel Capitolo 10.

Riepilogo

In questo capitolo sono stati presentati i componenti centrali del linguaggio Java. Con gli importanti miglioramenti apportati ai punti deboli del C e del C++, Java è destinato a sostituire questi due linguaggi in futuro. Gli elementi trattati in questo capitolo sono soltanto la punta dell'iceberg rappresentato dai vantaggi della programmazione in Java.

Nel prossimo capitolo sono trattati espressioni, operatori e strutture di controllo, e si mettono in pratica molte delle nozioni apprese in questo capitolo. Si inizia a scrivere programmi che non si limitano a visualizzare frasi di saluto sullo schermo.